



# Test Driven Development (TDD)



1



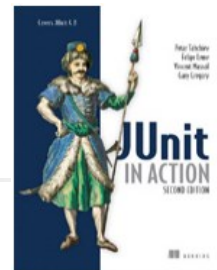
## Introduction

**How often have you heard this:**

**"We wanted to write tests, but we were under pressure and didn't have enough time to do it"**

**or;**

**"We started writing unit tests, but after two weeks our momentum dropped, and over time we stopped writing them."**



## Advantages of TDD

- Writing tests first require you to really *consider what you want from the code*
- Allows the *design* to evolve and adapt to your changing understanding of the problem.
- Short *feedback* loop
- Creates a *detailed specification*
- *Reduced time* in rework
- *Less time spent* in the *debugger* and when it is required you usually get closer to problem quickly

## Advantages of TDD

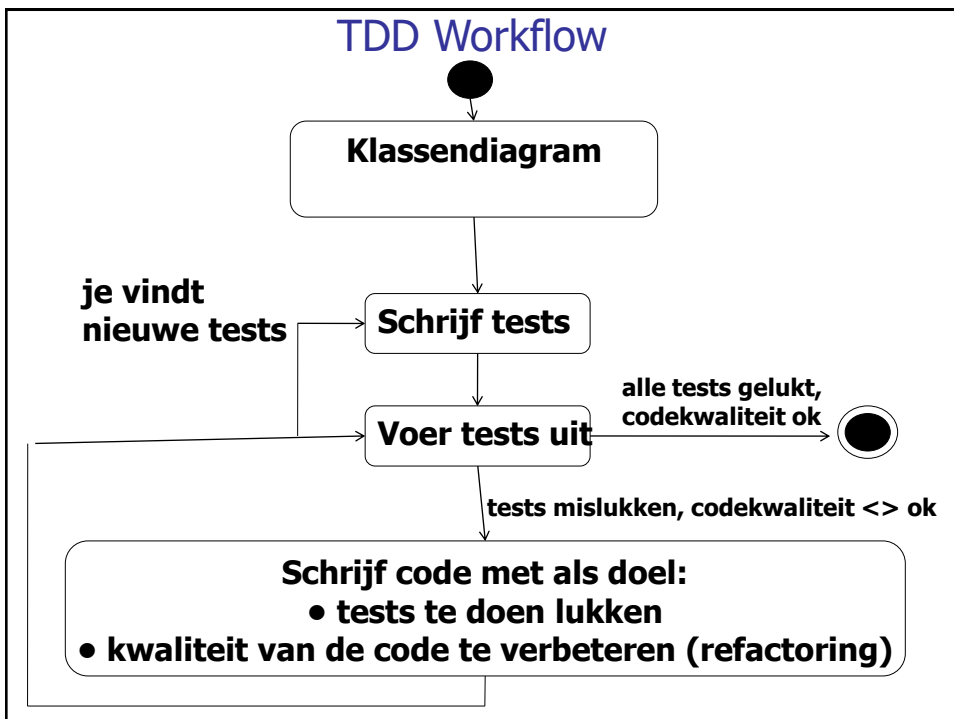
- Tells you whether your *last change (or refactoring)* has broken previously working code
- *Simplification:*
  - Forces radical simplification of the code – you will only write code in response to the requirements of the tests.
  - Forces you to write small classes focused on one thing.
  - Helps create loosely coupled code.
  - Creates SOLID code.

## Advantages of TDD

- The resulting Unit Tests are *simple* and act as *documentation* for the code
- *Improves quality and reduces bugs* by:
  - forcing us to consider purpose (and the specification of code)
  - simplifying code (many bugs come from the complexity)
  - ensuring changes and new code don't break the expectations of existing code

<http://agilepainrelief.com/notesfromatooluser/2008/10/advantages-of-tdd.html>

## TDD Workflow





## Test First Programming

---

- Je schrijf tests op functionaliteit, **vóór** je die functionaliteit uitcodeert.
- Bij het schrijven van de tests denk je na over de te coderen functionaliteit, op basis van de analyse.
- Code waarover je vooraf nadenkt, is **betere code**.
- Je voert deze tests uit  
Alle tests moeten falen, gezien je de functionaliteit nog niet uitwerkte.



## Test First Programming

---

- Als je tests schrijft nadat je functionaliteit codeert, heb je de neiging deze tests te baseren op de code van de functionaliteit, niet op de oorspronkelijke analyse.

**Je maakt dan regelmatig dezelfde denkfouten in je test als in je code.**



## Test First Programming

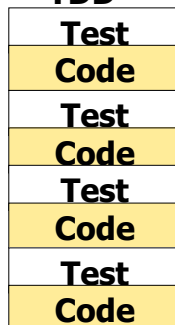
Door eerst de test te schrijven, **kijk je** naar een klasse **als een gebruiker van een klasse**.

- Als je, op basis van de UML design van een klasse, moeilijk een test kan schrijven, is deze design verkeerd.
- Je krijgt dus **snel feedback** op de UML design van de klasse.



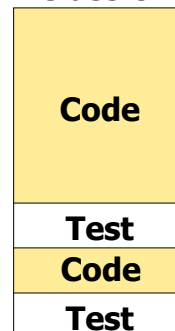
## TDD t.o.v. classic development

### TDD



Je schrijf een stukje functionaliteit en test dit ...  
Je krijgt continu feedback.

### Classic

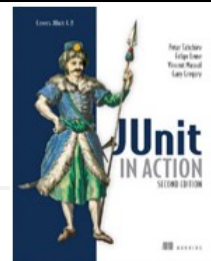


Je schrijf eerst veel functionaliteit.  
Daarna test je pas.  
Je krijgt lange tijd geen feedback.



- **Unit testing framework (open source)**
- **Ontworpen door Kent Beck en Erich Gamma in 1997**
- **"Standard for unit testing Java applications"**
- **JUnit 4**

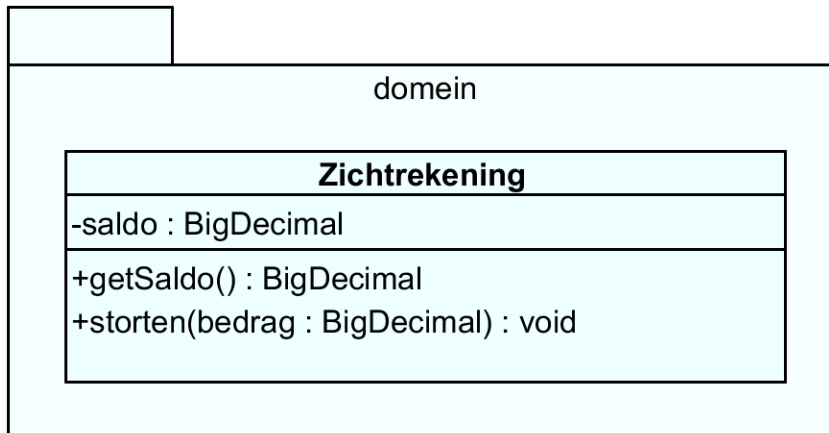
11



- **The framework must help us write useful tests.**
- **The framework must help us create tests that retain their value over time.**
- **The framework must help us lower the cost of writing tests by reusing code.**

12

## Vb: Design van de te testen klasse



13

## Hoe schrijven we een test voor deze klasse?

- Je schrijft de test **vóór** je een klasse implementeert.
- Je kan vanuit deze test methoden van de klasse niet oproepen als deze methoden niet bestaan.
- Je schrijft de methoden met een minimum aan code
  - Maak de klasse
  - Schrijf in de methode geen functionele code, maar werp een **UnsupportedOperationException**

14



## De te testen klasse

```
package domein;
import java.math.BigDecimal;

public class Zichtrekening
{
    private BigDecimal saldo;

    public BigDecimal getSaldo() {
        return this.saldo;
    }

    public void storten (BigDecimal bedrag) {
        throw new UnsupportedOperationException();
    }
}
```

15



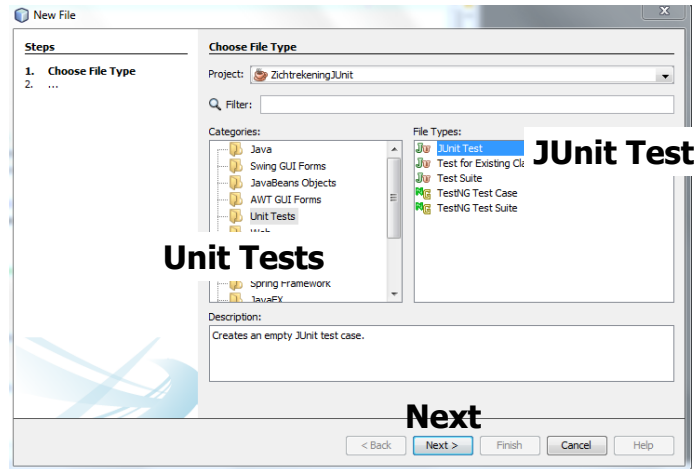
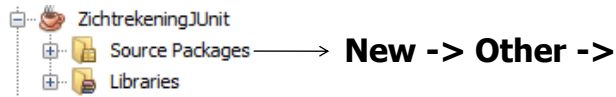
## JUNIT Test Case

- Test Case: een klasse die tests bevat
- Een Test Case is een POJO klasse:
  - Moet niet afgeleid zijn van een JUnit klasse
  - Moet geen JUnit interface implementeren
- JUnit aanziet iedere methode van een Test Case die voorzien is van **@Test** als een methode die iets test.
- Normale flow in een test methode:
  1. Maak een object van de te testen klasse
  2. Roep de te testen methode op
  3. Controleer of die methode correct uitgevoerd werd

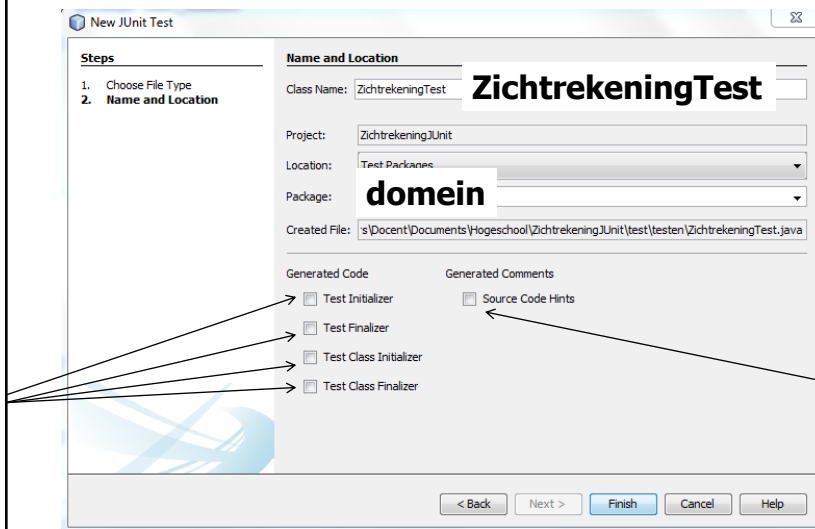
16



# Netbeans: JUNIT Test

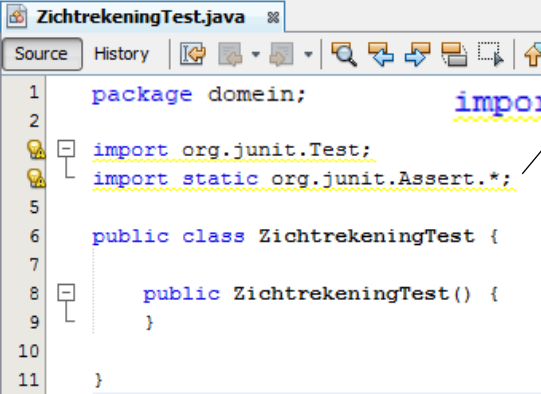


17



18

**wijzigen**



```
1 package domein;
2 import org.junit.Assert;
3 import org.junit.Test;
4 import static org.junit.Assert.*;
5
6 public class ZichtrekeningTest {
7
8     public ZichtrekeningTest() {
9     }
10
11 }
```

19



## Testscenario's

- 1) Jan heeft een nieuwe zichtrekening.  
Het saldo is gelijk aan 0.**
- 2) Jan heeft een nieuwe zichtrekening.  
Hij stort 200 euro op zijn rekening. Het saldo  
wordt dan 200 euro.**

```
package domein; //Test Packages
import java.math.BigDecimal;
import org.junit.Assert;
import org.junit.Test;
```

## JUNIT Test Case

```
public class ZichtrekeningTest {
```

```
    @Test
```

```
    public void stortenMoetSaldoAanpassen() {
        BigDecimal bedrag = new BigDecimal(200);
        Zichtrekening rekening = new Zichtrekening();
        rekening.storten(bedrag);
        Assert.assertEquals(bedrag, rekening.getSaldo());
    }
```

```
    @Test
```

```
    public void nieuweRekeningGeeftSaldoNul() {
        Zichtrekening rekening = new Zichtrekening();
        Assert.assertEquals(BigDecimal.ZERO,
            rekening.getSaldo());
    }
```

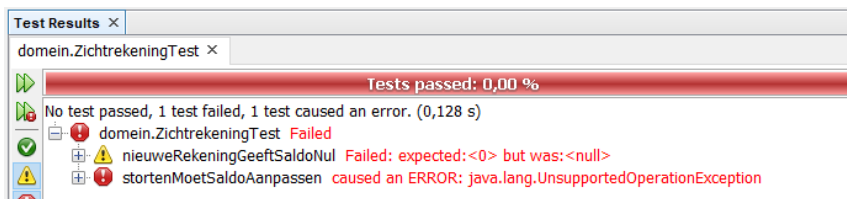
```
}
```

21



## JUNIT Test Case uitvoeren

### Run File



- **Error**  
Exception die optreedt in de te testen class
- **Failure**  
Assert methode die false teruggeeft

22

# De te testen klasse



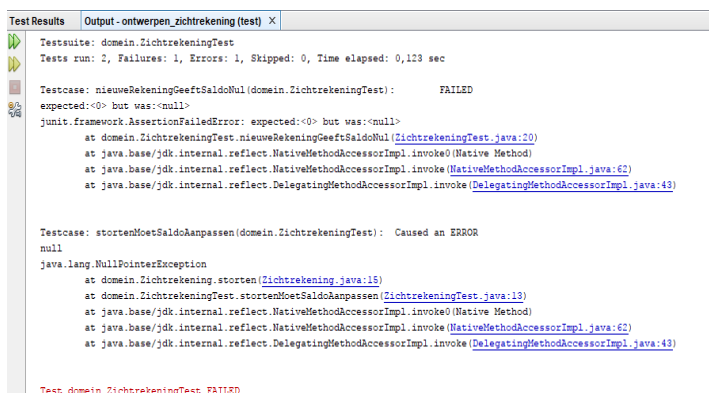
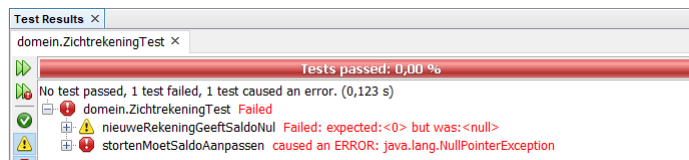
```
package domein;  
import java.math.BigDecimal;  
  
public class Zichtrekening  
{  
    private BigDecimal saldo;  
  
    public BigDecimal getSaldo()  
    {  
        return saldo;  
    }  
  
    public void storten (BigDecimal bedrag)  
    {//throw new UnsupportedOperationException();  
        saldo=saldo.add(bedrag);  
    }  
}
```

23

## JUNIT Test Case uitvoeren



### Run File



menu:

Window ->  
Output

24

## De klasse verbeteren



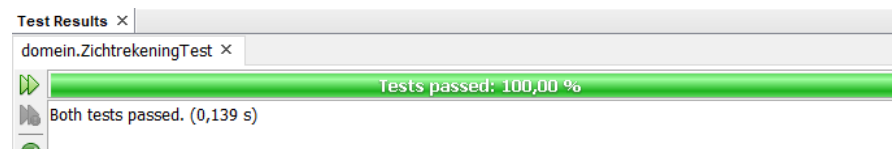
```
package domein;  
import java.math.BigDecimal;  
  
public class Zichtrekening  
{  
    private BigDecimal saldo = BigDecimal.ZERO;  
  
    public BigDecimal getSaldo()  
    {  
        return saldo;  
    }  
  
    public void starten (BigDecimal bedrag)  
    {  
        saldo=saldo.add(bedrag);  
    }  
}
```

25

## JUNIT Test Case uitvoeren



### Run File



“Keep the bar **green** to keep your code clean.”

26

## Static-methoden in klasse Assert



- **assertEquals(expected, actual)**  
Test of expected gelijk is aan actual.  
expected en actual kunnen als type hebben:
  - Alle primitieve datatypes
  - Object (en derived)Dan voert assertEquals expected.equals(actual) uit.
- **assertEquals(String message, expected, actual)**  
Test of expected gelijk is aan actual,  
Toont message als dit niet zo is.
- **assertEquals(expected, actual, delta)**  
Test of expected actual benadert.  
delta = maximaal verschil  
expected, actual en delta zijn van het type double of float.

## Static-methoden in klasse Assert



- **assertFalse(boolean conditie)**  
Test of conditie gelijk is aan false.
- **assertFalse(String message, boolean conditie)**  
Test of conditie gelijk is aan false.  
Toont message als dit niet zo is.
- **assertTrue(boolean conditie)**  
Test of conditie gelijk is aan true.
- **assertTrue(String message, boolean conditie)**  
Test of conditie gelijk is aan true.  
Toont message als dit niet zo is.

## Static-methoden in klasse Assert



- **assertNull(Object object)**  
Test of object gelijk is aan null.
- **assertNull(String message, Object object)**  
Test of object gelijk is aan null.  
Toont message als dit niet zo is.
- **assertNotNull(Object object)**  
Test of object verschillend is van null.
- **assertNotNull(String message, Object object)**  
Test of object verschillend is van null.  
Toont message als dit niet zo is.
- ...

29



## @Test methoden tips

- **Test niet te veel in één @Test methode**  
Schrijf dus niet te veel asserts in één @Test methode
- **Nadelen van veel testen in één @Test methode:**
  - Als een assert methode binnen een @Test methode faalt, voert JUnit de rest van die @Test methode niet uit
  - De methode wordt moeilijk te lezen
  - De kans dat je in de methode een bug schrijft is groot
  - De kans dat je de methode moet debuggen is groot

30



## @Test methoden tips

- De werking van één @Test methode mag **niet afhangen** van de werking van een andere @Test methode
- Je moet @Test methoden **in willekeurige volgorde** kunnen uitvoeren (JUnit garandeert geen volgorde van het uitvoeren van de @Test methods)

31



## @Test methode: slecht voorbeeld

```
...
public class ZichtrekeningTest {
    // slecht:alle @Test methode herbruiken de variable rekening
    // en zijn dus van mekaar afhankelijk !
    private static Zichtrekening rekening = new Zichtrekening();

    @Test
    public void stortenMoetSaldoAanpassen() {
        BigDecimal bedrag = new BigDecimal(200);
        rekening.storten(bedrag);
        Assert.assertEquals(bedrag, rekening.getSaldo());
    }
    @Test
    public void nieuweRekeningGeeftSaldoNul() {
        Assert.assertEquals(BigDecimal.ZERO, rekening.getSaldo());
    }
}
```

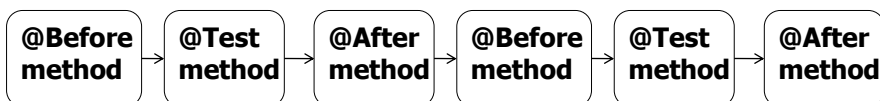
32





## Test fixture

- **Test fixture:**  
object of resource die je nodig hebt in meerdere @Test methods
- Declareer de variabele om naar de test fixture te verwijzen als private in de Test Case
- Initialiseer de test fixture in een methode voorzien van **@Before**
- JUnit voert de **@Before** methode uit vóór iedere **@Test** methode
- Kuis de resource op in een methode voorzien van **@After**
- JUnit voert deze methode uit na iedere **@Test** methode



33



## Test fixture

```
public class ZichtrekeningTest {  
    private Zichtrekening rekening;  
  
    @Before  
    public void before() // de naam van de methode mag je zelf kiezen  
    {  
        rekening = new Zichtrekening();  
    }  
  
    @Test  
    public void stortenMoetSaldoAanpassen() {  
        BigDecimal bedrag = new BigDecimal(200);  
        rekening.storten(bedrag);  
        Assert.assertEquals(bedrag, rekening.getSaldo());  
    }  
}
```

34



## Import static

```
import static org.junit.Assert.*;

...
public class ZichtrekeningTest {
    private Zichtrekening rekening;

    @Before
    public void before()
    { rekening = new Zichtrekening(); }

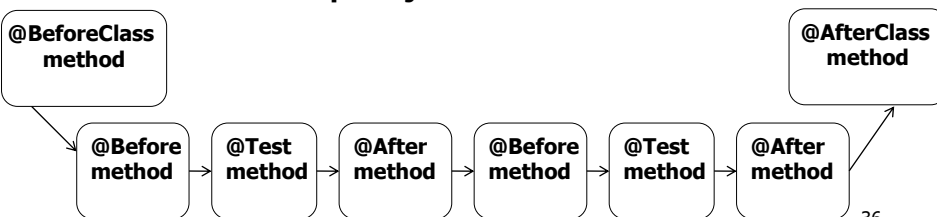
    @Test
    public void stortenMoetSaldoAanpassen() {
        BigDecimal bedrag = new BigDecimal(200);
        rekening.storten(bedrag);
        assertEquals(bedrag, rekening.getSaldo());
    }
    ...
}
```

35



## @BeforeClass en @AfterClass

- Als je een static methode voorziet van @BeforeClass, voert JUnit deze methode één keer uit, vóór alle @Test methods
- Je kan er test fixtures initialiseren
  - die de @Test methoden niet wijzigen
  - en veel tijd vragen om te initialiseren
- Als je een static methode voorziet van @AfterClass, voert JUnit deze methode één keer uit, na alle @Test methoden
- Kuis er de resources op dat je aanmaakte in @BeforeClass



36



## Wat te testen

- Grenswaarden uit de realiteit  
`rekening.storten(BigDecimal.ZERO);`
- Niet toegelaten waarden uit de realiteit  
`rekening.storten(new BigDecimal(-100));`  
`IllegalArgumentException` verwacht
- Niet toegelaten waarden (technisch)  
`rekening.storten(null);`  
`NullPointerException` verwacht

37

## Wat te testen (verzamelingen)

### GetallenStatistiek

<code>+min(getallen: int[]): int</code> <code>+max(getallen: int[]): int</code>
--

- Gewone verzameling:  
`getallenStatistiek.min(new int[] {10,20,30,40});`
- Verzameling met grenswaarden:  
`getallenStatistiek.min(  
  new int[] {Integer.MIN_VALUE, Integer.MAX_VALUE});`
- Verzameling met één element:  
`getallenStatistiek.min(new int[] {7});`
- Lege verzameling:  
`getallenStatistiek.min(new int[] {});`
- null:  
`getallenStatistiek.min(null);`

38



## Wat te testen (Strings)

Rekening
-rekeningnummer: String
+Rekening(rekeningnummer: String)

- Normale waarde:  
new Rekening("063-1547563-60");
- Lege string:  
new Rekening("");
- null  
new Rekening(null);
- te korte – te lange string  
new Rekening("063-1547563");  
new Rekening("063-1547563-601");
- string met vreemde tekens  
new Rekening("063é1547563@60");

39



## Testen op exceptions

- Als je verkeerde parameterwaarden meegeeft aan de methoden van de te testen klasse, moeten deze methoden Exceptions werpen.
- Je voorziet @Test van een parameter expected. Je vult deze parameter met een exception klasse.  
**@Test(expected = IllegalArgumentException.class)**
- Als de @Test methode geen exception werpt van deze exception klasse, krijg je een JUnit failure.

40

## Testen op exceptions: de tests

**@Test(expected = IllegalArgumentException.class)**

**public void stortenMetNegatiefBedrag()**

**{**

**BigDecimal bedrag = new BigDecimal(-10);**  
**rekening.storten(bedrag);**

**}**

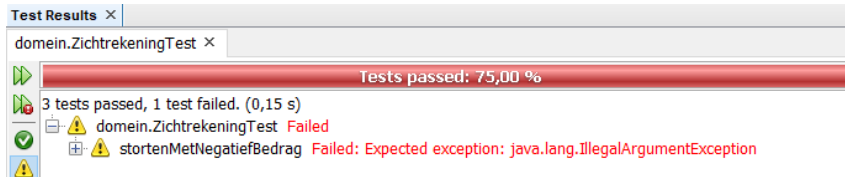
**@Test(expected = NullPointerException.class)**

**public void stortenMetNull()**

**{**

**rekening.storten(null);**

**}**



## Correctie in de te testen klasse

**public void storten(BigDecimal bedrag) {**

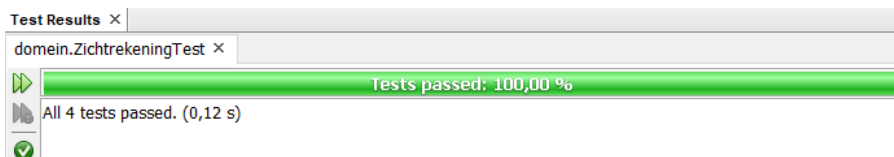
**if (bedrag.signum() < 0)**

**/\* OF if (bedrag.compareTo(BigDecimal.ZERO) < 0)**

**throw new IllegalArgumentException(**  
**"negatief bedrag kan niet gestort worden");**

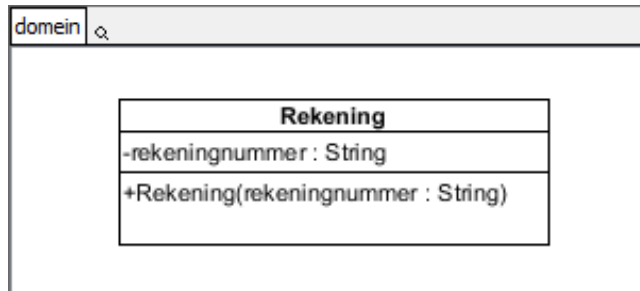
**saldo = saldo.add(bedrag);**

**}**





## Oefening



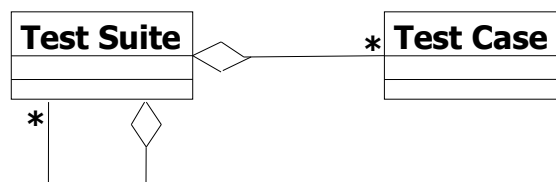
**Formaat: 3cijfers-7cijfers-2cijfers**  
 De rest van de deling eerste tien cijfers door 97 moet gelijk zijn aan de twee laatste cijfers.

43



## JUnit: Test Case - Test Suite

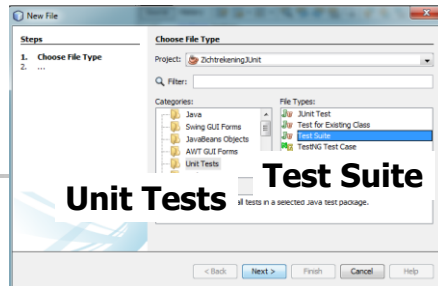
- **Test Case**  
 Klasse met **@Test** methode
- **Test Suite**  
 Verzameling van:
  - Test cases
  - Test suites



44

## New File

# Test Suite



**package domein; // Test Packages**

```
import org.junit.runner.RunWith;  
import org.junit.runners.Suite;
```

```
@RunWith(Suite.class)  
@Suite.SuiteClasses(  
    {domein.ZichtrekeningTest.class,  
    domein.RekeningTest.class } )  
public class AllTests {  
}
```

45



# Parameterized Test

- Een test meerdere keer uitvoeren met verschillende parameters

**Bv:** We gaan de volgende klasse testen:

```
package domain;  
  
public class Calculator  
{  
    public double add(double a, double b)  
    {  
        return a+b;  
    }  
}
```

46

## Parameterized Test



- We voorzien enkele testgevallen:

$1 + 1 = 2$

$2,5 + 1 = 3,5$

$3,1 + 1,1 = 4,2$

```
package domein;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
...
@RunWith(value = Parameterized.class)
public class ParameterizedTest
{
    private double expected;
    private double valueOne;
    private double valueTwo;
    private Calculator calculator;
```

47



### @Parameters

```
public static Collection<Double[]> getTestParameters()
{
    return Arrays.asList(
        new Double[][] { { 2.0, 1.0, 1.0 }, { 3.5, 2.5, 1.0 },
                          { 4.2, 3.1, 1.1 } });
}

public ParameterizedTest(double expected, double
valueOne, double valueTwo)
{
    this.expected = expected;
    this.valueOne = valueOne;
    this.valueTwo = valueTwo;
}
```

48



```
@Before
public void before()
{
    this. calculator = new Calculator();
}

@Test
public void sum()
{
    Assert.assertEquals(expected,
        calculator.add(valueOne, valueTwo), 0);
}
}
```

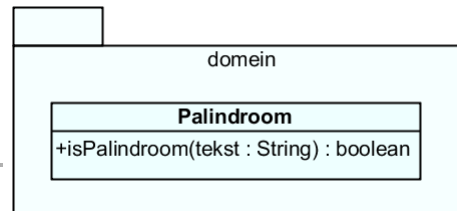


## Oefening

**Herschrijf de klasse RekeningTest, door gebruik te maken van 'Parameterized Test'**



## Oefening



**Een palindroom is een symmetrische sequentie, zoals een woord of een getal.**

**Voorbeelden van palindroomgetallen zijn 1001 en 12345678987654321: van links naar rechts gelezen zijn ze hetzelfde als van rechts naar links gelezen.**

**Voorbeelden van palindromische woorden zijn woorden in de Nederlandse taal zoals kok, pap, lol, lepel etc.**

51



## Oefening



**Voeg de Test Case "PalindroomTest" toe aan de Test Suite.**

52